

Laboratorio Arquitectura de Computadoras

Santiago Ramirez Arenas

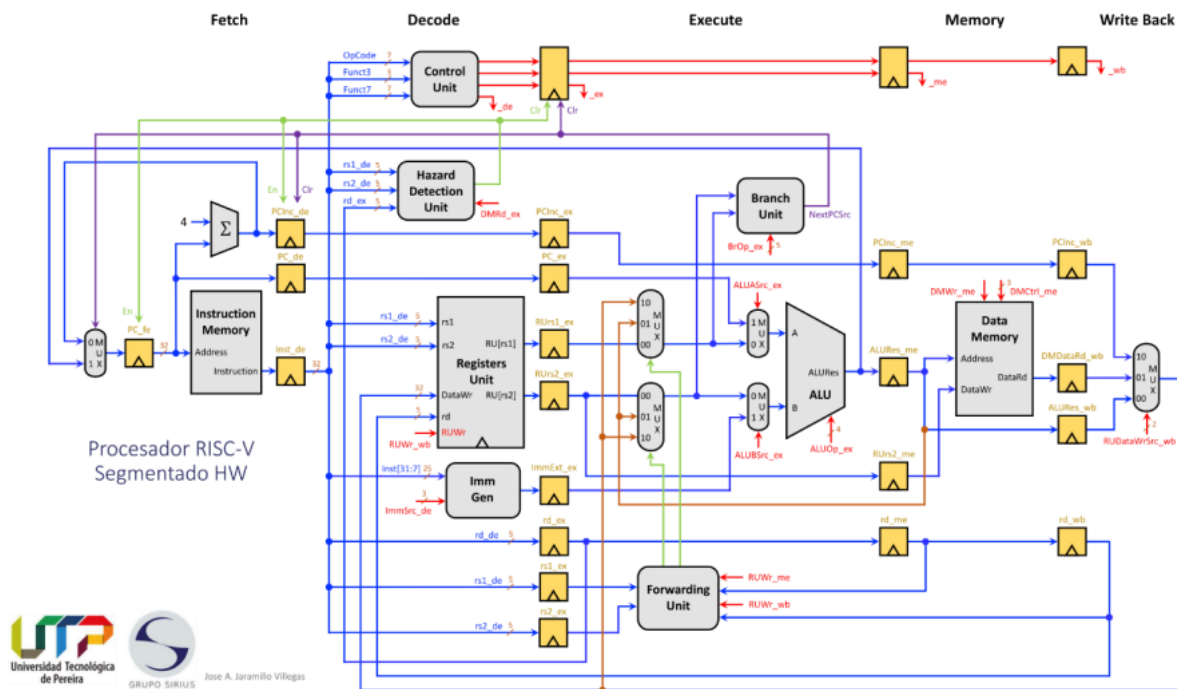
Docente: José Alfredo Jaramillo Villegas

Universidad Tecnológica de Pereira

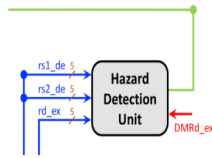
11 de diciembre de 2021

Procesador Segmentado.

Este tipo de procesador tiene como principio 5 etapas: Fetching, Decoding, Execution, Memory Accessing y Write Back. Cuenta con varios módulos que logran darle la capacidad de procesamiento los cuales están representados en el gráfico.



Hazard Unit

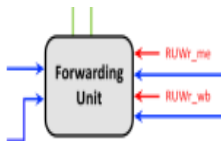


```

1 module hazard_unit(
2     assign Flush = (ForceJump_EX || ForceJump_MEM || ForceJump_WB);
3     assign FlushPC = (ForceJump_ID || ForceJump_EX);
4     assign FlushAddr = (ForceJump_ID || ForceJump_EX || ForceJump_MEM)
5
6     always_comb
7     if ((MemRead_EX == 1'b1') && (rd_addr_EX == rs1_addr_ID) || (rd_addr_EX
8         == rs2_addr_ID)) && !BranchMux_MEM)
9         begin
10             PCWrite_ID = 1'b0;
11             HazardMux = 1'b1;
12         end
13     else begin
14         PCWrite_UD = 1'b1;
15         HazardMux = 1'b0;
16     end
17 endmodule )
18

```

Forwarding Unit



```

1 module Forwarding_unit (
2     always_comb
3     begin
4         if((RegWrite_MEM==1) && (rd_addr_MEM!='0') &&
5             (rs1_addr_EX==rd_addr_MEM))
6             Forward1=2'b10;
7         else if ((RegWrite_WB==1) && (rd_addr_WB!='0') &&
8             (rs1_addr_EX==rd_addr_WB))
9             Forward1=2'b01;
10        else
11            Forward1=2'b00;
12
13        if((RegWrite_MEM==1) && (rd_addr_MEM!='0') &&
14            (rs2_addr_EX==rd_addr_MEM))
15            Forward2=2'b10;
16        else if ((RegWrite_WB==1) && (rd_addr_WB!='0')
17            && (rs2_addr_EX==rd_addr_WB))
18            Forward2=2'b01;
19    end
20

```

```

15         else
16             Forward2=2'b00;
17     end
18 endmodule )
19
20

```

Diseño Proceador Segmentado

```

1  'include "instruction_memory.sv"
2  'include "registers_unit.sv"
3  'include "ALU.sv"
4  'include "data_memory.sv"
5  'include "imm_gen.sv"
6  'include "branch_unit.sv"
7  'include "control_unit.sv"
8  'include "program_counter.sv"
9  'include "adder.sv"
10 'include "MUX2x1.sv"
11 'include "MUX3x1.sv"
12
13 //Fetching
14 'include "PC_fe.sv"
15
16 //Decoder
17 'include "PCInc_de.sv"
18 'include "PC_de.sv"
19 'include "Inst_de.sv"
20
21 //Execution
22 'include "PCInc_ex.sv"
23 'include "PC_ex.sv"
24 'include "RUrs1_ex.sv"
25 'include "RUrs2_ex.sv"
26 'include "ImmExt_ex.sv"
27 'include "Rd_ex.sv"
28
29 //Memory
30 'include "PCInc_me.sv"
31 'include "ALURes_me.sv"
32 'include "RUrs2_me.sv"
33 'include "Rd_me.sv"
34
35 //Write Back
36 'include "PCInc_wb.sv"
37 'include "DMDataRd_wb.sv"
38 'include "ALURes_wb.sv"
39 'include "Rd_wb.sv"
40
41
42
43 module procesador_segmentado (input CLK);
44     wire CLK;
45     wire [31:0] PC_Out;
46     wire [31:0] adder_Out;

```

```

47 wire [31:0] Instruction;
48 wire [1:0] RUDataWrSrc;
49 wire ALUASrc, ALUBSrc;
50 wire [31:0] Data1;
51 wire [31:0] Data2;
52 wire RUWr;
53 wire [2:0] ImmSrc;
54 wire [31:0] ImmExt;
55 wire [4:0] BrOp;
56 wire NextPCSrc;
57 wire [31:0] ALURes;
58 wire [3:0] ALUOp;
59 wire [31:0] DataRd;
60 wire DMWr;
61 wire [2:0] DMCtrl;
62 wire [31:0] Out_mux3x1;
63 wire [31:0] Out_pc_mux;
64 wire [31:0] ALUMUXResA;
65 wire [31:0] ALUMUXResB;
66
67 program_counter pc (
68     .CLK (CLK,)
69     .PC_IN (out_pc_mux),
70     .PC_Out (PC_Out));
71
72 adder add (
73     .adder_In (PC_Out),
74     .adder_Out (adder_Out));
75
76 instruction_memory im (
77     .Address (PC_out),
78     . Instruction (Instruction));
79
80 control_unit cu (
81     .OpCode (Instruction [6:0]),
82     .Funct3 (Instruction [14:12]),
83     .Funct7 (Instruction [31:25]),
84     .ALUASrc (ALUASrc),
85     .ALUBSrc (ALUBSrc),
86     .ALUOp (ALUOp),
87     .ImmSrc (ImmSrc),
88     .DMWr (DMWr),
89     .DMCtrl (DMCtrl),
90     .RUDataWrSrc (RUDataWrSrc),
91     .RUWr (RUWr),
92     .BrOp (BrOp));
93
94 registers_unit ru (
95     .rs1 (Instruction [19:15]),
96     .rs2 (Instruction [24:20]),
97     .rd (Instruction [11:7]),
98     .DataWr (Out_mux3x1),
99     .RUWr (RUWr),
100    .CLK (CLK),

```

```

101         .Data1 (Data1),
102         .Data2 (Data2));
103
104     imm_gen ig (
105         .Inst (Instruction),
106         .ImmSrc (ImmSrc),
107         .ImmExt (ImmExt));
108
109     branch_unit bu (
110         .A (Data1),
111         .B (Data2),
112         .BrOp (BrOp),
113         .NextPCSrc (NextPCSrc));
114
115     MUX2x1 alu_mux_a (
116         .In1 (Data1),
117         .In2 (PC_Out),
118         .Src (ALUASrc),
119         .Out (ALUMUXResA));
120
121     MUX2x1 alu_mux_b (
122         .In1 (Data2),
123         .In2 (ImmExt),
124         .Src (ALUBSrc),
125         .Out (ALUMUXResB));
126
127     ALU alu (
128         .A (ALUMUXResA),
129         .B (ALUMUXResB),
130         .ALUOp (ALUOp),
131         .ALURes (ALURes));
132
133     data_memory dm (
134         .Address (ALURes),
135         .DataWr (Data2),
136         .DMWr (DMWr),
137         .DMCtrl (DMCtrl),
138         .DataRd (DataRd));
139
140     MUX3x1 mux3to1(
141         .In1 (ALURes),
142         .In2 (DataRd),
143         .In3 (adder_Out),
144         .RUDataWrSrc (RUDataWeSrc),
145         .Out (Out_mux3x1));
146
147     MUX2x1 pc_mux(
148         .In1 (adder_Out),
149         .In2 (ALURes),
150         .Src (NextPCSrc),
151         .Out (Out_pc_mux));
152
153     //FETCHING
154

```

```

155 PC_FE PC_fe(
156     .CLK(clk),
157     .In(pc),
158     .Out(OUTpc_fe)
159 );
160
161 //DECODER
162
163 PCINC_DE PCInc_de(
164     .CLK(clk),
165     .In(bsum),
166     .Out(OUTpcinc_de)
167 );
168
169 PC_DE PC_de(
170     .CLK(clk),
171     .In(instruction),
172     .Out(OUTpc_de)
173 );
174
175 INST_DE INST_de(
176     .CLK(clk),
177     .In(instruction),
178     .Out(OUTinst_de)
179 );
180
181
182 //EXECUTE
183
184 PC_EX PC_ex(
185     .CLK(clk),
186     .In(OUTpc_de),
187     .Out(outpc_ex)
188 );
189
190 RD_EX RD_ex(
191     .CLK(clk),
192     .In(OUTinst_de),
193     .Out(OUTrd_ex)
194 );
195
196 PCINC_EX PCInc_ex(
197     .CLK(clk),
198     .In(OUTpcinc_de),
199     .Out(OUTpcinc_ex)
200 );
201
202 RURS1_EX RUrs1_ex(
203     .CLK(clk),
204     .In(rus1),
205     .Out(OUTrurs1_ex)
206 );
207
208 RURS2_EX RUrs2_ex(

```

```

209     .CLK(clk),
210     .In(rus2),
211     .Out(OUTrurs2_ex)
212 );
213
214 IMMEXT_EX IMMext_ex(
215     .CLK(clk),
216     .In(immext),
217     .Out(OUTimmext_ex)
218 );
219
220
221 //MEMORY
222 PCINC_ME PCinc_me(
223     .CLK(clk),
224     .In(OUTpcinc_ex),
225     .Out(OUTpcinc_me)
226 );
227
228 RD_ME RD_me(
229     .CLK(clk),
230     .In(OUTrd_ex),
231     .Out(OUTrd_me)
232 );
233
234 RURS2_ME RUrs2_me(
235     .CLK(clk),
236     .In(OUTrurs2_ex),
237     .Out(OUTrurs2_me)
238 );
239
240 ALURES_ME ALUres_me(
241     .CLK(clk),
242     .In(alures),
243     .Out(OUTalures_me)
244 );
245
246 //WRITE BACK
247 RD_WB RD_wb(
248     .CLK(clk),
249     .In(OUTrd_me),
250     .Out(outrd_wb)
251 );
252
253 PCINC_WB PCInc_wb(
254     .CLK(clk),
255     .In(OUTpcinc_me),
256     .Out(outpcinc_wb)
257 );
258
259 DMDATARD_WB DMDataRd_wb(
260     .CLK(clk),
261     .In(datamemory),
262     .Out(OUTdmdataard_wb)

```

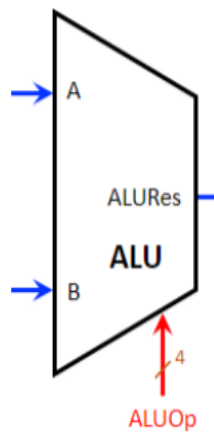
```

263 );
264
265 ALURES_WB ALUres_wb(
266     .CLK(clk),
267     .In(OUTalures_me),
268     .Out(OUTalures_wb)
269 );
270
271 endmodule

```

Unidad aritmecológica (ALU):

Creación de una ALU de dos operandos (A y B tamaño de 32 bits), un ALUOp (tamaño de 4 bits) que asigna una operación y devuelve el resultado (tamaño de 32 bits).



```

1 module ALU (
2     input logic signed [31:0] A,
3     input logic signed [31:0] B,
4     input logic [3:0] ALUOp,
5     output logic signed [31:0] ALURes);
6
7     always @ (*)
8     begin
9         case (ALUOp)
10            4'b0000:
11                ALURes <= A + B;
12            4'b1000:
13                ALURes <= A - B;
14            4'b0001:
15                ALURes <= A << B;
16            4'b0010:
17                ALURes <= A < B;
18            4'b0011:
19                ALURes <= $unsigned (A);
20            4'b0100:
21                ALURes <= A ^ B;
22            4'b0101:
23                ALURes <= A >> B;

```



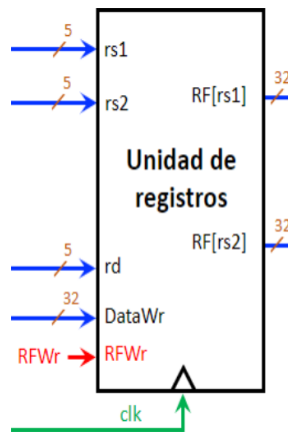
```

24         4'b1101:
25             ALURes <= A >>> B;
26         4'b0110:
27             ALURes <= A | B;
28         4'b0111:
29             ALURes <= A & B;
30         4'b1111:
31             ALURes <= A <= B;
32     endcase
33 end
34 endmodule

```

- Unidad de registros:

Creación del módulo Unidad de Registros. El cual cuenta con los siguientes registros (cada uno con su respectivo número de bits): el ingreso de la dirección de dos registros(rs1, rs2), dos salidas con la información que fue solicitada (RF[rs1], RF[rs2]), activador de lectura (rd), activador de escritura(RFWr) e información de registro para ingresar(DataWrz).



```

1 module registers_unit (
2     input logic [4:0] rs1,
3     input logic [4:0] rs2,
4     input logic [4:0] rd,
5     input logic [31:0] DataWr,
6     input logic RUWr,
7     input logic CLK,
8     output logic [31:0] Data1,
9     output logic [31:0] Data2);
10
11     logic [31:0] RU [31:0];
12
13     initial
14         $readmemb ("registers.txt",RU);
15
16     assign Data1 = RU [rs1];
17     assign Data2 = RU [rs2];
18

```

```

19  always @ (posedge CLK)
20      begin
21          if (RUWr == 1 && rd != 5'b0)
22              RU [rd] <= DataWr;
23              $monitor ("x20 value: %d", RU [20])
24      end
25 endmodule

```

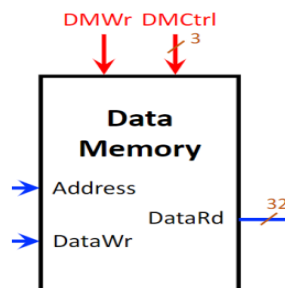
Memoria de datos.

Se define como una matriz en la cual se puede almacenar y leer los datos que van a ser procesados. Este módulo del procesador se encarga de almacenar todos los datos que van a perdurar y, a su vez, recuperar los datos solicitados. El módulo cuenta con los siguientes registros de entrada y salida (cada uno con su respectivo número de bits):

Escritura: La dirección de ingreso para almacenar (Address), el valor que se quiere almacenar (DataWr) y la bandera que activa la escritura (DMWr).

Lectura: Tipo de lectura DMCtrl (Byte, Halfword, Word, Unsigned Byte, Unsigned Halfword), La dirección de ingreso para leer la memoria (Address).

Nota: Address funciona para los casos (lectura y escritura)



```

1  module data_memory (
2      input logic [31:0] Address,
3      input logic signed [31:0] DataWr,
4      input logic DMWr,
5      input logic [2:0] DMCtrl,
6      output logic signed [31:0] DataRd );
7
8      integer 1;
9      parameter memory_size 2**20;
10     logic [7:0] Memory [Memory_size -1: 0];
11
12     //Escritura
13     always @ (*)
14     begin
15         if (DMWr == 1'b1)
16             begin
17                 case (DMCtrl)

```

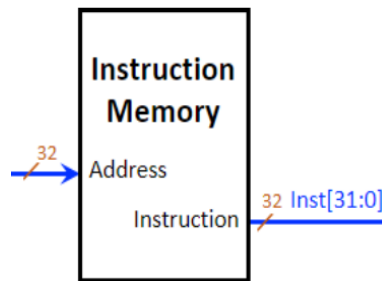
```

18         3'b000: // Byte
19             Memory [Address] <= DataWr [7:0];
20         3'b001: // Media palabra
21             begin
22                 Memory [Address] <= DataWr [7:0];
23                 Memory [Address + 1] <= DataWr [15:8];
24             end
25         3'b010: // Palabra
26             begin
27                 for (i = 0; i < 4; i = i + 1)
28                     Memory [Address + 1] <= DataWr [i*8 + :8];
29             end
30         endcase
31     end
32 end
33
34 //Lectura
35 always @ (*)
36     begin
37         case (DMCtrl)
38         3'b000: // Byte
39             DataRd <= {{24{Memory[Address][7]}}, Memory [Address]};
40         3'b001: // Media palabra
41             DataRd <= {{16{Memory[Address + 1][7]}}, Memory [Address + 1],
Memory [Address]};
42         3'b010: // Palabra
43             DataRd <= {{Memory[Address + 3], Memory [Address + 2], Memory
[Address + 1], Memory[Address]};
44         3'b100: //Byte unsigned
45             DataRd <= {24'b0, Memory [Address]};
46         3'b101 // Media palabra unsigned
47             DataRd <= {16'b0, Memory [Address + 1], Memory[Address]};
48             default:
49                 DataRd = 31'bx;
50         endcase
51     end
52 endmodule

```

Memoria de instrucciones.

Se puede definir como una matriz en la cual se almacenan y se leen las instrucciones que se van a ejecutar. Este módulo del procesador recibe un Address el cual es la dirección de la instrucción que se quiere recuperar y una salida Instruction la cual es la instrucción que fue recuperada de la matriz



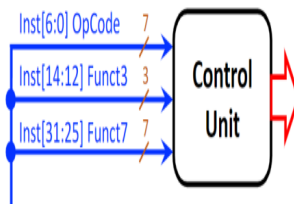
```

1 module instruction_memory (
2     input logic [31:0] Address,
3     input logic [31:0] Instruction);
4
5     parameter memory_size = 2 ** 20;
6
7     logic [7:0] Memory [memory_size - 1: 0];
8
9     initial begin
10         $readmemh ("instructions.txt", Memory);
11     end
12
13     always @ (*)
14         Instruction <= {Memory [Address + 3], Memory [Address + 2],
15             Memory [Address + 1], Memory [Address]};
16 endmodule

```

Unidad de control

Es la encargada de decodificar la instrucción de entrada (OpCode, Funct3, Funct7) para generar cada una de las señales de control (RUWr, ALUASrc, ALUBSrc, ALUOp, BrOp, DMWr, DMCtrl, RUDataWrSrc e ImmSrc).



```

1 module control_unit (
2     input logic [6:0].OpCode,
3     input logic [2:0] Funct3,
4     input logic [6:0] Funct7,
5     output logic ALUASrc,
6     output logic ALUBSrc,
7     output logic [3:0] ALUOp,
8     output logic [2:0] ImmSrc,
9     output logic DMWr,
10    output logic [2:0] DMCtrl,
11    output logic [1:0] RUDataWeSrc,
12    output logic RUWr,
13    output logic [4:0] BrOp;
14
15    always @ (*)
16        begin
17            case (OpCode)
18                7'b0110011; //Instrucion tipo R
19                begin
20                    ALUASrc = 1'b0;
21                    ALUBSrc = 1'b0;
22                    ALUOp = {Funct7 [7], Funct3};
23                    ImmSrc = 3'bxxx;
24                    DMWr = 1'b0;
25                    CMCtrl = 3'bxxx;
26                    RUDataWrSrc = 2'b00;
27                    RUWr = 1'b1;
28                    BrOp = 5'b00xx;
29                end
30
31                7'b0010011: // Instruccion tipo I
32                begin
33                    ALUASrc = 1'b0;
34                    ALUBSrc = 1'b1;
35                    if (Funct3 == 3'b101)
36                        ALUOp = {Funct7 [5], Funct3};
37                    else
38                        ALUOp = {1'b0, Funct3};
39                    ImmSrc = 3'b000;
40                    DMWr = 1'b0;
41                    DMCtrl = 3'bxxx;
42                    RUDataWeSrc = 2'b00;
43                    RUWr = 1'b1;
44                    BrOp = 5'b00xx;
45                end
46
47                7'b0000011: // Instruccion tipo I load
48                begin
49                    ALUASrc = 1'b0;
50                    ALUBSrc = 1'b1;
51                    ALUOp = 4'b0000;
52                    ImmSrc = 3'b000;
53                    DMWr = 1'b0;
54                    DMCtrl = Funct3;

```

```

55         RUDataWrSrc = 2'b01;
56         RUWr = 1'b1;
57         BrOp = 5'b00xxx;
58     end
59
60     7'b1100111: // Instruccion tipo I jalr
61     begin
62         ALUASrc = 1'b0;
63         ALUBSrc = 1'b1;
64         ALUOp = 4'b0000;
65         ImmSrc = 3'b000;
66         DMWr = 1'b0;
67         DMCtrl = 3'bxxx;
68         RUDataWrSrc = 2'b10;
69         RUWr = 1'b0;
70         BrOp = 5'b1xxxx;
71     end
72
73     7'b1100011: //Instruccion tipo B
74     begin
75         ALUASrc = 1'b1;
76         ALUBSrc = 1'b1;
77         ALUOp = 4'b0000;
78         ImmSrc = 3'b101;
79         DMWr = 1'b0;
80         DMCtrl = 3'bxxx;
81         RUDataWrSrc = 2'bxx;
82         RUWr = 1'b0;
83         BrOp = {2'b01, Funct3};
84     end
85
86     7'b0100011: //Instruccion tipo S
87     begin
88         ALUASrc = 0;
89         ALUBSrc = 1;
90         ALUOp = 4'b0000;
91         ImmSrc = 3'b001;
92         DMWr = 1'b1;
93         DMCtrl = Funct3;
94         RUDataWrSrc = 2'bxx;
95         RUWr = 1'b0;
96         BrOp = 5'b00xx;
97     end
98
99     7'b1101111: //Instruccion tipo J -Jal
100    begin
101        ALUASrc = 1'b1;
102        ALUBSrc = 1'b1;
103        ALUOp = 4'b0000;
104        ImmSrc = 3'b110;
105        DMWr = 1'b0;
106        DMCtrl = 3'bxxx;
107        RUDataWrSrc = 2'b10;
108        RUWr = 1'b1;

```

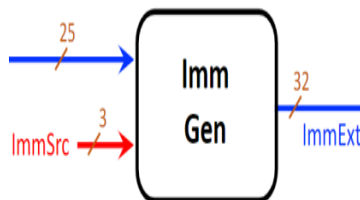
```

109         BrOp = 5'b1xxxx;
110     end
111
112     7'b0110111: //Instruccion tipo U -lui
113     begin
114         ALUASrc = 1'bx;
115         ALUBSrc = 1'b1;
116         ALUOp = 4'b1111;
117         ImmSrc = 3'b010;
118         DMWr = 1'b0;
119         DMCtrl = 3'bxxx;
120         RUDataWrSrc = 2'b00;
121         RUWr = 1'b1;
122         BrOp = 5'b100xxx;
123     end
124
125     7'b0110111: //Instruccion tipo U -auipc
126     begin
127         ALUASrc = 1'b1;
128         ALUBSrc = 1'b1;
129         ALUOp = 4'b0000;
130         ImmSrc = 3'b010;
131         DMWr = 1'b0;
132         DMCtrl = 3'bxxx;
133         RUDataWrSrc = 2'b00;
134         RUWr = 1'b1;
135         BrOp = 5'b100xxx;
136     end
137 endcase
138 end
139 endmodule

```

Generador de immediatos

El generador de immediatos es el encargado de ensamblar el inmediato decodificado en la instrucción de entrada según el tipo de esta



```

1 module imm_gen(
2     input logic [31:0] Inst,
3     input logic [2:0] ImmSrc,
4     output logic [31:0] ImmExt);
5
6     always @ (*)
7     begin
8         case (ImmSrc)

```

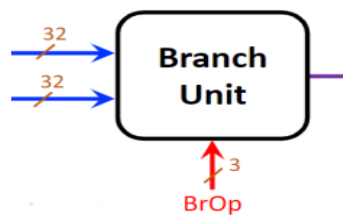
```

9      3'b000: //Instruccion tipo I
10         ImmExt = {{20{Inst{31}}}}, Inst {31:20}};
11      3'b001: //Instruccion tipo S
12         ImmExt = {{20{Inst{31}}}}, Inst {31:25}, Inst{11:7}};
13      3'b101: //Instruccion tipo B
14         ImmExt = {{19{Inst{31}}}}, Inst {31}, Inst{7},Inst {30:25},Inst
15         {11:8}, 1'b0};
16      3'b010: //Instruccion tipo U
17         ImmExt = {{Inst{Inst{31:12}}}}, 12'b0};
18      3'b110: //Instruccion tipo J
19         ImmExt = {{19{Inst{31}}}}, Inst {31}, Inst{19:12},Inst
20         {20},Inst {30:21}, 1'b0};
21         default;
22         ImmExt = 31'bx;
23     endcase
24 end
25 endmodule

```

Branch unit

Esta se encarga de determinar si la instrucción de entrada ordena un salto en la dirección de la memoria de instrucciones



```

1 module branch_unit(
2     input logic signed  [31:0] A, //Es asignado de RU [rs1]
3     input logic signed  [31:0] B, //Es asignado de RU [rs2]
4     input logic  [4:0] BrOp,
5     output logic NextPCSrc );
6
7     always @ (*)
8     begin
9         if (BrOp[4])
10             NextPCSrc = 1;
11         else
12             if (BrOp [3])
13                 case (BrOp[2:0])
14                     3'b000; //BEQ
15                     NextPCSrc = (A == B);
16                     3'b001; //BNE
17                     NextPCSrc = (A != B);
18                     3'b100; //BLT
19                     NextPCSrc = (A < B);
20                     3'b101; //BGE
21                     NextPCSrc = (A >= B);
22                     3'b110; //BLTU

```



```

23         NextPCSrc = ($unsigned (A) < $unsigned (B));
24         3'b111; //BGEU
25         NextPCSrc = ($unsigned (A) >= $unsigned (B));
26         default:
27             NextPCSrc = 1'bx;
28     endcase
29 else
30     NextPCSrc = 0;
31 end
32 endmodule
33

```

Program counter

```

1 module program_counter (
2     input logic CLK,
3     input logic [31:0] PC_In,
4     output logic [31:0] PC_Out = 0 );
5
6     always @ (posedge CLK)
7         PC_Out <= PC_In;
8 endmodule

```

Adder

```

1 module adder (
2     input [31:0] adder_In,
3     output [31:0] adder_Out );
4
5     assign adder_Out = adder_In + 4;
6 endmodule

```

MUX 2x1

```

1 module MUX2x1 (
2     input logic [31:0] In1,
3     input logic [31:0] In2,
4     input logic Src,
5     output logic [31:0] Out);
6
7     always @ (*)
8         begin
9             case (Src)
10                1'b0:
11                    Out <= In1;
12                1'b1:
13                    Out <= In2;
14            endcase
15        end
16 endmodule

```

MUX 3x1

```

1 module MUX3x1 (
2     input logic [31:0] In1,
3     input logic [31:0] In2,
4     input logic [31:0] In3,

```

```

5  input logic  [1:0]RUDataWrSrc ,
6  output logic [31:0]Out);
7
8  always @ (*)
9      begin
10         case (RUDataWeSrc)
11             2'b00:
12                 Out <= In1;
13             2'b01:
14                 Out <= In2;
15             2'b10:
16                 Out <= In3;
17         endcase
18     end
19 endmodule

```

Testbench

```

1 module testbench;
2     logic CLK;
3
4     procesador_segmentado pm (CLK)
5
6     always
7         #5
8         CLK = ~CLK;
9     initial
10        begin
11            $dumpfile ("dump.vcd");
12            $dumpvars();
13            CLK=0;
14            #850
15            $finish;
16        end
17 endmodule

```

Instructions

```

1  FF010113
2  00812623
3  00912423
4  01212223
5  01312023
6  00050413
7  00000913
8  00000993
9  00000993
10 0099D863
11 00890933
12 00198993
13 FE000AE3
14 00090513
15 00C12403
16 00812483
17 00412903
18 00012983

```

```
19 01010113
20 00008067
21 00040513
22 0FC000EF
23 00050493
24 FF010113
25 00812623
26 00912423
27 01212223
28 00112023
29 00050413
30 00100913
31 00140413
32 00100493
33 0084DE63
34 00090513
35 00048593
36 DD1FF0EF
37 00050913
38 00148493
39 FE0004E3
40 00090513
41 00C12403
42 00812483
43 00412903
44 00012083
45 00012083
46 01010113
47 00008067
```

Registers

```
1 00000000000000000000000000000000
2 00000000000000000000000000000000
3 00000000000000000000000000000000
4 00000000000000000000000000000000
5 00000000000000000000000000000000
6 00000000000000000000000000000000
7 00000000000000000000000000000000
8 00000000000000000000000000000000
9 00000000000000000000000000000000
10 00000000000000000000000000000000
11 00000000000000000000000000000000
12 00000000000000000000000000000000
13 00000000000000000000000000000000
14 00000000000000000000000000000000
15 00000000000000000000000000000000
16 00000000000000000000000000000000
17 00000000000000000000000000000000
18 00000000000000000000000000000000
19 00000000000000000000000000000000
20 00000000000000000000000000000000
21 00000000000000000000000000000000
22 00000000000000000000000000000000
23 00000000000000000000000000000000
```

```

24 00000000000000000000000000000000
25 00000000000000000000000000000000
26 00000000000000000000000000000000
27 00000000000000000000000000000000
28 00000000000000000000000000000000
29 00000000000000000000000000000000
30 00000000000000000000000000000000
31 00000000000000000000000000000000
32 00000000000000000000000000000000

```

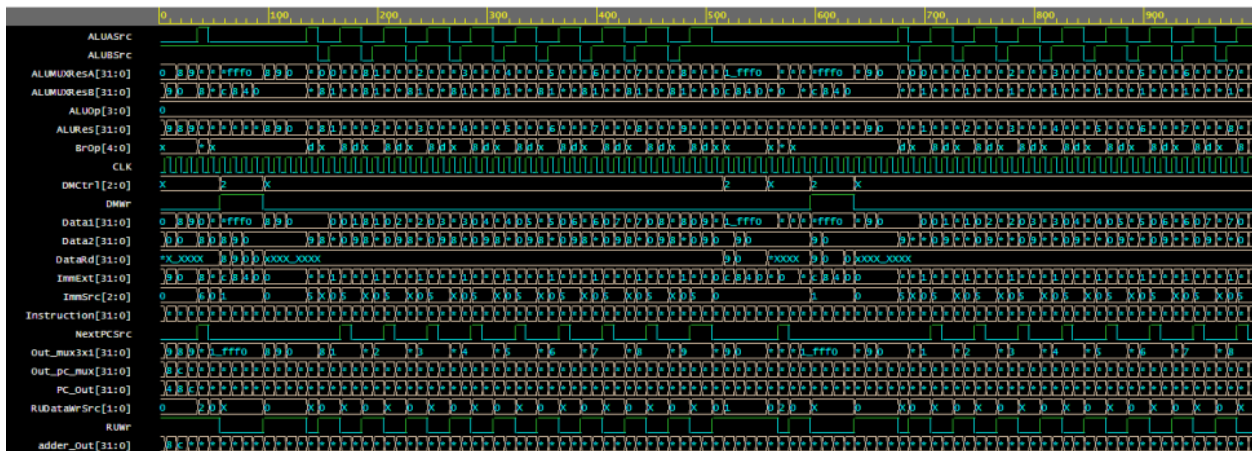
Resultado

```

1 #KERNEL: x20 value 0
2 #KERNEL: x20 value 72
3 RUNTIME: Info: RUNTIME_0068 testbemch.sv (16): $finish calle.
4 #KERNEL: Time: 1 us, Iteration: 0, Instance: / testbench, Process:
  @INITIAL#19_1@.
5 #KERNEL: stopped at time: 1 us
6 #VSIM: Simulation has finished. There are no more test vectors to
  simulate.
7 #VSIM: Simulation has finished

```

Simulación



Instrucciones RV32I

Tipo R				Tipo I			Tipo S B		
	OpCode	Funct3	Funct7		OpCode	Funct3		OpCode	Funct3
add	0110011	000	0000000	addi	0010011	000	beq	1100011	000
sub	0110011	000	0100000	slli	0010011	001	bne	1100011	001
sll	0110011	001	0000000	slti	0010011	010	blt	1100011	100
slt	0110011	010	0000000	sltiu	0010011	011	bge	1100011	101
sltu	0110011	011	0000000	xori	0010011	100	bltu	1100011	110
xor	0110011	100	0000000	srai	0010011	101	bgeu	1100011	111
srl	0110011	101	0000000	ori	0010011	110	sb	0100011	000
sra	0110011	101	0100000	andi	0010011	111	sh	0100011	001
or	0110011	110	0000000				sw	0100011	010
and	0110011	111	0000000						

Tipo U J		
	OpCode	
jal	1101111	
lui	0110111	
auipc	0010111	

lb	0000011	000
lh	0000011	001
lw	0000011	010
lbu	0000011	100
lhu	0000011	101
jalr	1100111	000

Jose A. Jaramillo Villegas



Crear un código en C ++, llevar a ensamblador y por último a lenguaje de máquina. El cual será utilizado como las instrucciones de prueba para demostrar el correcto funcionamiento.

```

1  Funcion en C++
2
3  o Funcion en C++
4
5  int multi(int x, int y)
6  {
7      int Acc = 0;
8      for(int i = 0; i < y; i++) Acc += x;
9      return Acc;
10 }
11
12 int factorial(int num)
13 {
14     int fact = 1;
15     int num += 1;
16     for(int i = 1; i < num; i++) fact = multi(fact,i); return fact;
17 }
18
19 o Funcion en lenguaje ensamblador
20
21 Prototipo de la funcion multi:
22 x10      x      x8 x11      y      x9
23 Acc      x18
24 i      x19 multi(x,y)      x10
25
26 Prototipo de la funcion factorial: x10      num      x8
27 i      x9
28 fact      x18 factorial(num)      x10
    
```

```

29
30 Llamado a la función factorial: f = factorial(num)
31 num      x8
32 f        x9

```

Procedimiento del laboratorio:

Proceso realizado para el diseño del ejercicio.

Para el diseño del procesador monociclo RISC-V se implementó teniendo en cuenta las instrucciones suministradas y componentes que fueron proporcionados en el gráfico de dicho procesador, primero se declaraban teniendo en cuenta y agregando como argumentos las variables de entrada y salida, luego se describía el funcionamiento interno de cada módulo.

A continuación se concetaron los modulos y se distanciaron los módulos de cada proceso, además de se asignaron las instrucciones pertinentes para verificar el funcionamiento correcto de cada módulo y su conexión con módulos posteriores.

Problemas encontrados durante el diseño con sus respectivas las soluciones.

Se encontraron varios problemas en relación al distanciamiento de los módulos y la conexión entre varios componentes, así como algunas implementaciones en las cuales era necesario saber cierta sintaxis del lenguaje para describir el funcionamiento de dichos componentes, consultando en internet y entrando en foros se logró dar con la solución, además de la documentación que fue proporcionada en el Microsft Teams.

Interpretación de los resultados de la simulación.

En la simulación se puede observar la señal de cada uno de los componentes y como se conectaban entre si, para dar lugar al resultado de una operación matemática, se verificó el funcionamiento haciendo manualmente la intrucción y posteriormente se verificaron los resultados

Conclusiones

RISC es una filosofía de diseño de CPU para computadora que está a favor de conjuntos de instrucciones pequeñas y simples que toman menor tiempo para ejecutarse. El tipo de procesador más comúnmente utilizado en equipos de escritorio, el x86, está basado en CISC en lugar de RISC, aunque las versiones más nuevas traducen instrucciones basadas en CISC x86 a instrucciones más simples basadas en RISC para uso interno antes de su ejecución.

El objetivo de diseñar máquinas la arquitectura RISC FIVE es posibilitar la segmentación y el paralelismo en la ejecución de instrucciones y reducir los accesos a memoria. Las máquinas RISC protagonizan la tendencia actual de construcción de microprocesadores. PowerPC,2 DEC Alpha, MIPS, ARM, SPARC son ejemplos de algunos de ellos